Characterizing a New Class of Threads in Scientific Applications for High End Supercomputers

Arun Rodrigues⋆ Richard Murphy⋆ Peter Kogge⋆ Keith Underwood‡

University of Notre Dame⋆
Computer Science and Engineering Department
384 Fitzpatrick Hall
Notre Dame, IN 46545

Sandia National Lab[*] ‡
PO Box 5800
MS-1110
Albuquerque, NM 87185-1110

{arodrig6,rcm,kogge}@cse.nd.edu, kdunder@sandia.gov

Abstract

Chip level multithreading is growing in use throughout the microprocessor world as evidenced in the Intel Pentium 4 and the upcoming innovations in the POWER architecture. These processors typically use a few coarse grain threads that can be difficult for the programmer or compiler to exploit; however, Processing in Memory (PIM) is a technology that has been explored through a long series of supercomputer projects as a facilitator for a different multithreaded execution models. In the multithreading model explored by PIMs, the threads can have radically different characteristics. Specifically, PIMs seek to exploit a large number of very fine grained threads to hide memory access latency and increase parallelism. PIM supports these small threads, or "threadlets", by providing a fast hardware synchronization mechanism, support for harware managment of creation and destruction of threads, and a "shared register" approach which extends the shared memory thread model. This paper discusses some analysis of some very large scientific codes in terms of how they might be mapped onto such a multithreading model with a focus on extremely fine grain threads.

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

1. Introduction

Multithreading is growing in popularity as a mechanism to increase the utilization of processor resources. The Intel Pentium 4 and the upcoming Power5 architectures include some support for multithreading so that one thread may continue while another thread waits on memory. The success of these techniques depends on the ability to provide a small number of loosely synchronized threads to execute and to provide adequate bandwidth to feed data to those threads. One aspect of the Cray Cascade project, which is part of the DARPA HPCS program, is to consider Processing-In-Memory (PIM) technology that drastically increases the memory bandwidth available, dramatically decreases the granularity of the threads executed, and provides fine grain synchronization between the threads.

PIM technology [18, 19, 10, 23, 22] (also known as IRAM [25], merged memory and logic [11], etc) , as discussed here, involves a significant change in the way systems are built. Instead of using separate components for processing and memory, PIM assumes one. Large amounts of processing logic are placed on the same die as dense memory macros, and together form a single part type modular computer building block. The first such parts that implemented multiple complete CPU+memory nodes on a single chip are now over a decade old [17, 33] and a great many alternative architectures geared explicitly for such technology have been proposed [9, 24, 12] and some built or in the process of being built, such as the Mitsubishi MRD32, Micron Yukon, DAMM, and Blue Gene/C[5].

Some novel execution models have been proposed for PIM that involve new variants of multithreading, where threads are not tied to a single CPU, but instead are allowed to be initiated and executed on processing logic *in the vicinity of* the data being processed. Examples of such novel models include the work that led to the parcel model of the HTMT and DIVA projects [32, 15], the concept of a *microserver* as a PIM node *memory-resident* manager of remote initiations [10], and of threads that can move from memory to memory [22].

Given the growing interest in multithreading technology and the potential for the emergence of a supercomputer architecture that supports extremely fine grained threads (referred to as “threadlets” in this paper), a number of questions emerge. For example: what exactly are the thread-type *metaphors* that might be applicable to such models, how often do they appear in real codes, and what are their characteristics. This paper provides some early answers to these questions, and lays the foundation for ongoing experiments into novel PIM-supported microarchitectural design points. Further, it does so not in the context of SPEC-like benchmarks, but in terms of some very large scientific codes. These codes represent the workloads that run on modern supercomputers and that are expected to run on supercomputers in the near future.

In terms of organization, the next section presents some multithreading background and a definition of the types of threads discussed in this work. Section 3 addresses the thread extraction process used in the analysis, and the key parameters that were measured. Section 4 discusses the applications and some of their basic characteristics. Section 5 then addresses the overall results in terms of thread specific characteristics. Finally, Section 6 outlines the key observations that are derived from this, and lays the path for continuing work in the area.

2. Threadlets

Multithreading is a natural match for the type of latency tolerance required by a PIM system. It allows for concurrency to be discovered by the programmer and compiler, rather than the hardware. It also allows the latency of memory accesses and pipelining to be masked by the available threads, rather than by more complex hardware-oriented schemes. In contrast, most multithreaded architectures focus on a small number of relatively large threads that are used to increase total throughput, rather than to mask memory latency. For example, Intel’s Hyperthreading [21] or the multithreaded PowerPC[8] support precisely this type of server-oriented workload. The Cray (Tera) MTA[6], on the other hand, has explored using large numbers of relatively small threads, with extremely light weight fork, join, and synchronization mechanisms.

This paper focuses on the opportunities for parallelism in very tiny threads or *threadlets* with fine grain support for thread creation, destruction, and synchronization. This is similar to other parallelism limit studies[31, 20], except that it is targeted towards fine-grain multithreaded PIM systems, in which the threads are primarily identified by the compiler. Threadlets exist within a *basic block*, which is a sequence of instructions occurring between two branch instructions in the program trace. This restriction serves to eliminate

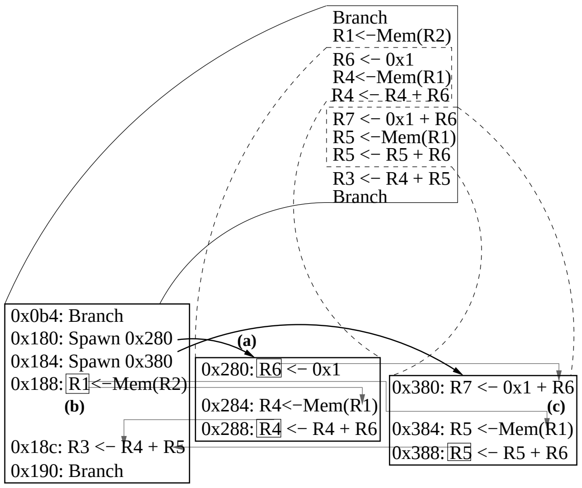

Figure 1. A thread spawns two threadlets (a), each of size 3. Synchronization is required whenever a value is produced by one thread (b) and consumed by another (c).

control dependencies. The threadlet consists of the original set of instructions (generated by a typical compiler) along with fork and join operations (signifying the beginning and end of a threadlet) and a set of synchronizations (Figure 1). Synchronizations are required whenever one threadlet produces (Figure 1(b)) a value which another threadlet consumes (Figure 1(c)). This value may be stored to a memory location by one threadlet and then loaded by another, or it may be an integer or floating point register value computed by one threadlet and then used by another. To ensure correctness, there must be a synchronization mechanism to ensure that the consuming threadlet has some way of knowing if the producing threadlet has not yet produced the required value so it can wait for the required data. Thus, threadlets serve to extract potential concurrency from a serial program. By examining a program trace, we are able to quantify all the opportunities for threadlet creation as a motivation for compiler writers. Furthermore, the characteristics of threadlets identify the critical architectural features required for their execution.

To efficiently support low-level threading, a fast hardware supported synchronization mechanism must be provided. One such mechanism is the Full/Empty Bit (FEB), as used in the Cray MTA[13] or the Sparcle [4] processors. FEBs attach an extra bit of state to each word in memory to designate if that word is “full” or “empty.” A thread which produces a data value sets the word’s state to “full” when it stores the value. Consuming threads check the word’s state before completing a load. If the word is not full, the consuming thread will block. Because this mechanism is supported by hardware and accessing a word’s status is an atomic operation with the data access this mechanism can be very low cost.

FEBs can also be included on registers[34] allowing multiple threads to share the same register context. Sharing registers reduces the number of hardware register contexts required to support a given number of threads and it also reduces the costs of synchronization by not requiring data to be moved to and from memory. Shared register contexts is similar to the minithreads in the $_{mt}$SMT architecture [29].

To generate these threadlets, the program is transformed to consist of a large master thread, which forks off small threadlets opportunisticly (Figure 1(a)). These threads consist of perhaps half a dozen instructions, and may share state with larger threads (requiring programmer or compiler specified synchronization within the register file) and perform data driven synchronization (similar to the full/empty bit mechanism used in the MTA). Unlike the MTA, the actual threadlet state is extremely tiny (consisting minimally of a program counter and program status word).

The program trace is split into threadlets by constructing a data dependency graph, identifying the synchronization points, and applying the thread extraction algorithm given in Section 3. The dependency graph itself provides significant useful information about the potential concurrency available in the program: the graph’s *width* provides a measure of the maximum potential concurrency, and its *height* represents the critical path through the dependency graph (or shortest execution time).

One challenge of threadlets is that the relative number of required synchronizations increases with decreasing thread size. That is, groups of instructions which would normally have executed sequentially (within a thread) now require synchronization to coordinate. While the synchronization mechanism is provided by the architecture and is very low cost, for threads to make forward progress enough data must be available. Thus, this paper examines the characteristics of synchronization points between threads as well as the thread's general architectural characteristics.

3. Methodology

The goal of the threadlet extraction process is to take an instruction trace of a serial program and transform the trace into a parallel multithreaded trace in a realistic manner. Traces are gathered using the `amber` instruction trace generator for PowerPC[7]. These instruction traces were then converted to an architecture independent format called TT7 for further analysis. Data dependencies between instructions in the TT7 traces where identified, and a data dependency graph was generated (see Figure 2(a)). A "steering" partitioning scheme[35, 30], originally developed to distribute instructions between clusters in a multicluster architecture [14], was adapted to partition the data dependency graph into separate threadlets.

Once the data dependency graph has been generated threadlet extraction begins. To avoid dealing with control dependencies, threadlets are only extracted *within* a basic block. The trace is scanned again, and basic blocks are identified (Figure 2(b)). Partitioning is performed on each basic block as it is identified.

Each instruction in the basic block is analyzed to find an optimal threadlet assignment. The number of threadlets to extract from the block is defined by a variable. A parameter p is set to determine the target thread size. The heuristic attempts to extract as many threads of size p from the block as possible. For each threadlet, a "score" is computed based upon the number of parents (instructions which the current instruction depends on) already assigned to the threadlet (Figure 2(c)) and an estimated "load" for the threadlet, indicating the number of instructions that threadlet still needs to process (Figure 2(d)). The instruction is assigned to the threadlet with the best score, and the process begins again for the next instruction in the basic block.

This partitioning scheme attempts to minimize the need for synchronization between the threadlets, and achieve effective load balancing. To further reduce synchronization penalties, the heuristic "prefers" synchronizations between instructions which are further apart in program order. If an instruction must synchronize with another instruction which is earlier in program order, then there is a greater chance that the parent instruction will have already completed, and that the child threadlet will not have to block. For these reasons, the steering-based approach to partitioning tended to give better results than standard graph partitioning algorithms, such as those used in the METIS library [16].

4. Benchmarks

Scientific applications tend to be significantly different from common processor benchmarks. As such, it has become common in the course of supercomputer procurements to specify a set of applications as targets for the new machine. Examples of this include the ASCI Purple Benchmark Suite[1] and the "$7\times$" list for ASCI Red Storm[1]. This section describes three applications that were selected from the ASCI Red Storm "$7\times$" list as well as another significant consumer

[1]This is a list of 10 applications that are expected to run 7 times faster on ASCI Red Storm than on ASCI Red.

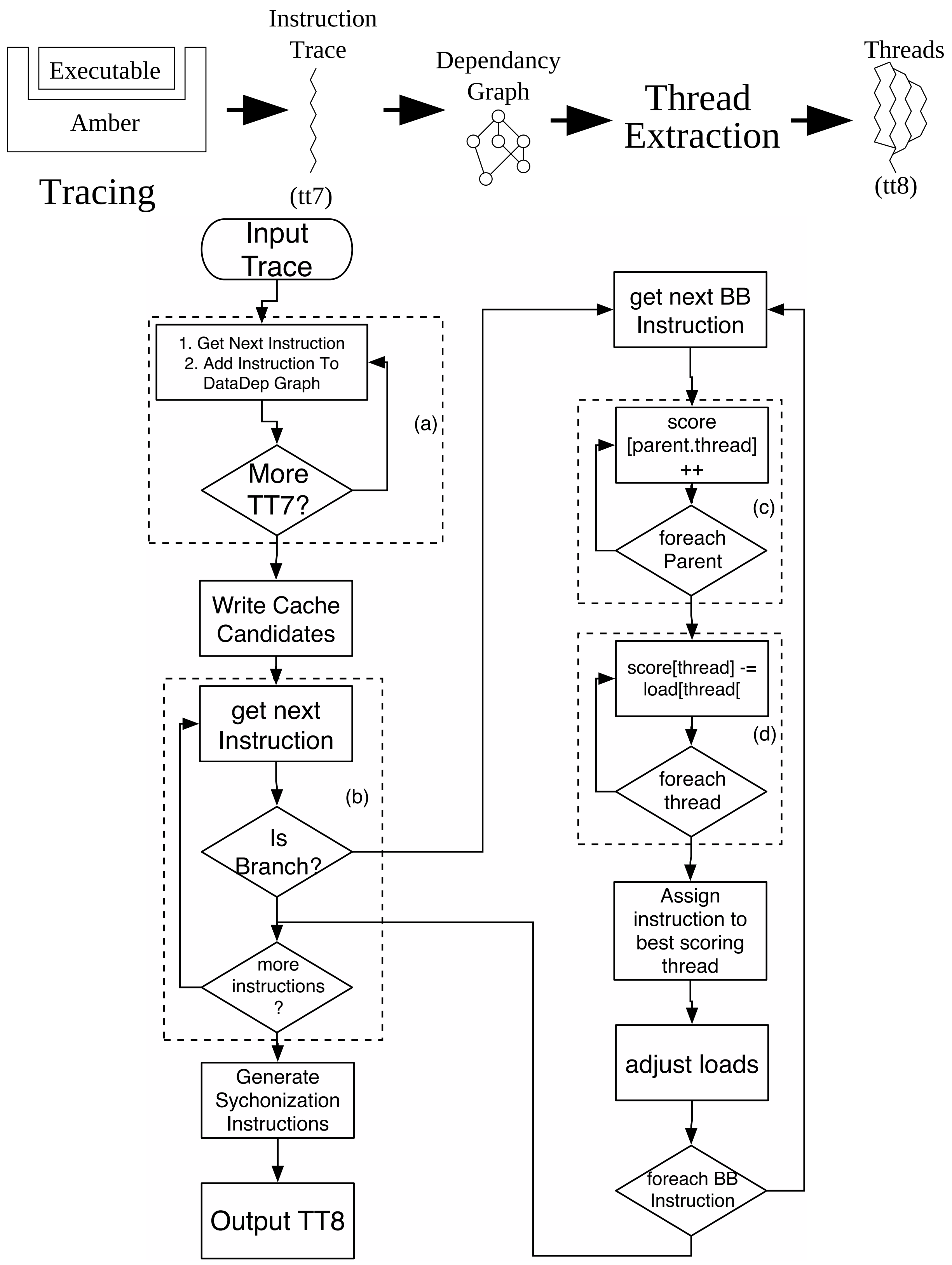

Figure 2. Trace Generation and Threadlet Extraction Process

of compute cycles at Sandia (LAMMPS). For each application, the input sets are also described.

4.1. LAMMPS

LAMMPS is a classical molecular dynamics (MD) code designed to simulate systems at the atomic or molecular level[27, 26, 28]. Typical applications include simulations of proteins in solution, liquid-crystals, polymers, zeolites, or simple Lenard-Jones systems. It runs on any parallel platform that supports the MPI message-passing library or on single-processor workstations. Version 2001 is written in Fortran90 and consists of approximately 30,000 lines of code[2].

In a typical simulation, some combination of the features of LAMMPS is used. Thus, four different input problems that demonstrated different characteristics were chosen for analysis. These were:

- *Lennard Jones Mixture with a baseline NVE ensemble* — This input simulated a 2048 atom system consisting of three different kinds of atoms running for 500 timesteps in an ensemble that holds the number of atoms (N), volume (V) and energy (E) constant.

- *Lennard Jones System with Induced Flow* — A small two dimensional simulation with 210 atoms pinned between two fixed walls. The upper wall moves to induce Couette flow in the mobile atoms. This uses the same kernel computation as the previous Lennard Jones simulation, but introduces new boundary conditions.

- *Bead-Spring Polymer Chains* — This is a simulation of a simple system with molecular bonds. Two types of idealized, 50-length, bead-spring polymer chains using different bead sizes are simulated along with some free monomers. The polymer chains first push off from each other for 10000 timesteps and then equilibrate for 10000 timesteps. The simulated system includes 810 atoms and runs for 20000 timesteps.

- *Liquid-Crystal Molecules* — A 6750 atom collection of 27 liquid-crystal molecules was simulated for 100 timesteps. Columbic forces were solved for via particle-mesh Ewald (an FFT-based solver). This simulation exercises many of the core computational routines in LAMMPS.

4.2. CTH

CTH is a multi-material, large deformation, strong shock wave, solid mechanics code developed at Sandia National Laboratories. CTH has models for multi-phase, elastic viscoplastic, porous and explosive materials. Three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes are available. It uses second-order accurate numerical methods to reduce dispersion and dissipation and produce accurate, efficient results. CTH is used extensively within the Department of Energy laboratory complexes for studying armor/anti-armor interactions, warhead design, high explosive initiation physics and weapons safety issues. It is implemented in Fortran and consists of approximately 500,000 lines of code.

CTH has two fundamental modes of operation: with or without adaptive mesh refinement (AMR). Adaptive mesh refinement changes the application properties significantly and is useful for only certain types of input problems. Therefore, we have chosen one AMR problem and one non-AMR problem for analysis. In addition, a third problem was chosen for analysis that significantly reduces the computational requirements for the equation of state. These test problems are:

[2]This text adapted with permission from http://www.cs.sandia.gov/ sjplimp/lammps.html.

- *CuSt* — This input problem simulates a 4.52 km/s impact of a 4 mm copper ball on a steel plate at a 90 degree angle. Adaptive mesh refinement is used in this problem.

- *Explosively Formed Projectile (EFP)* — The simulation represents a simple Explosively Formed Projectile (EFP) that was designed by SNL staff. The original design was a combined experimental and modeling activity where design changes were evaluated computationally before hardware was fabricated for testing. The design features a concave copper liner that is formed into an effective fragment by the focusing of shock waves from the detonation of the high explosive. The measured fragment size, shape, and velocity is accurately (within 5%) modeled by CTH.

- *2 Gas* — This problem uses an $80 \times 80 \times 80$ mesh. The simulated space contains two gases intersecting on a 45 degree plane. This problem uses a very simple equation of state.

4.3. ITS

The Integrated TIGER Series (ITS) is a suite of codes to perform Monte Carlo solutions of linear time-independent coupled electron/photon radiation transport problems. It can simulate problems with or without the presence of macroscopic electric and magnetic fields in multimaterial, multidimensional geometries. Individual particles are tracked with independent particle histories. Thus, particle transport is assumed to be a linear process in which individual particles do not interact with each other, or alter the medium in which they transport. The current version of the code supports two modes of geometry inputs: combinatorial geometry (CG) and CAD. The CG mode provides a custom input format for describing problem geometries as constructions of simple geometric primitives such as spheres, boxes, and cylinders. The CAD mode uses the ACIS library[2] to import problem descriptions directly from CAD drawings. The use of the ACIS library to extract problem geometries can significantly change the computational characteristics of the problem; thus, one problem was tested in both geometry modes. The ITS code base is approximately 66,000 lines of source code including both Fortran and C++.

The test problem used for this analysis models the Saturn x-ray source at Sandia National Labs. In some experiments with the Saturn facility, electron currents are generated and directed at a “converter”. This converter is composed of tantalum and aluminum components. The electrons interact with the tantalum via bremsstrahlung interactions that produce photons. Most of the photons escape the tantalum and pass through the aluminum and are used to irradiate the experimental device. The electrons, however, are unable to pass through the aluminum. The test problem used models this converter; thus, it models the conversion of electron radiation into photon radiation.

4.4. sPPM

The sPPM benchmark is part of the ASCI Purple benchmark suite[1] as well as the 7× application list for ASCI Red Storm. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) code. It represents the current state of ongoing research, which has demonstrated good processor performance, excellent multithreaded efficiency, and excellent message passing parallel speedups. The hydrodynamics algorithm involves a split scheme of X, Y, and Z Lagrangian and remap steps that are computed as three separate passes or sweeps through the mesh per timestep. Each sweep through the mesh requires approximately 680 FLOPs to update all of the state variables for each real mesh cell. The sPPM code is written in Fortran 77 with some C routines and contains a total of approximately

4000 lines of code.

The problem solved by the sPPM involves a shock propagating through a gas with a density discontinuity. A plane shock traveling up the +z axis encounters a density discontinuity, at which the gas becomes denser. The shock is carefully designed to be simple, though strong (about Mach 5). The gas initially has density 0.1 ahead of the shock; over 5dz at the discontinuity, it changes to 1.0. For a qualitative idea of the computation, see the volume renderings at [3][3].

4.5. Benchmark Comparison

Instruction traces were gathered for each of these benchmarks. Table 1 summarizes some key characteristics of the benchmark traces. **Trace Size** refers to the number of instructions captured for analysis. **% Traced (inst)** indicates the number of instructions captured as a percentage of all instructions executed. **% Traced (data)** indicates the number of 4K data pages touched in the captured trace as a percentage of the total number of data pages accessed by the full run of the program. **Mean Basic Block** is the arithmetic mean of the basic block sizes. **Unique Inst. pages** indicates the number of memory pages which contained instructions executed during the trace. **Unique Data pages** shows the number of pages containing data which was loaded or stored during the trace. For both **Unique Inst.** and **Unique Data pages**, a 4K page size was assumed.

These applications show several traits common to scientific codes. Most striking is the large basic block size. Unlike many conventional applications that have mean basic block lengths of only a few instructions, these benchmarks frequently have basic block sizes of over a dozen instructions. Other than ITS, basic block sizes range from 9.06 to over 20.

The applications also tend to be very data intensive, accessing hundreds to thousands of data pages and dozens of instruction pages. All of the applications, except sPPM, spend at least 40% of their instructions accessing memory (Figure 3). sPPM, being a benchmark and dealing with dense matrices, has somewhat lower memory requirements.

LAMMPS and CTH Integer ALU instructions tend to be dominated by addition, subtraction, multiplication and shifting operations (Figure 3). Divides are infrequent. ITS, having more branches, spends more time in comparison and logic operations operations. sPPM contains fewer integer ALU operations, and these tend to be mainly logical operations, most likely due to simplified address generation with dense matrices.

Floating point instructions are dominated by addition, subtraction and multiplications. In all but the CTH 2gas experiment these account for roughly 65% to 85% of floating point instructions (Figure 3). The CTH 2gas is also unique in the high number of floating point divides it performs - nearly 18%.

4.6. Graph Widths and Parallelism

A good measure of the potential parallelism available in these applications is be gained by analysis of the data dependence graphs of the trace. If instructions are nodes in this graph and data dependencies form the edges between them, we can find the average width of this graph by dividing the total number of instructions in the trace by the maximum height of the graph. This **graph width** tells us the average number of instructions which could be scheduled in parallel, assuming perfect knowledge of the program and perfect branch prediction. Another useful statistic is the width of the data dependency graph with control dependencies. To calculate this, we assume that all the instructions in one basic block must complete before instructions from the next basic block can begin executing. This has the effect of elongating the graph because instructions

[3]This text adapted with permission from http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/sppm.readme.html.

Benchmark	Abbrev	Trace Size	% Traced (inst)	% Traced (data)	Mean Basic Block	Unique Inst. pages	Unique Data pages
Dhrystone 1.1	NA	NA	NA	NA	4.28	NA	NA
gcc.122	NA	NA	NA	NA	5.37	NA	NA
LAMMPS lj nve	lmp.nve	420M	23%	100%	18.72	163	344
LAMMPS poly chain	lmp.poly	500M	12%	42%	21.79	45	118
LAMMPS big pppm	lmp.big	500M	3%	3%	16.76	54	124
LAMMPS langevin	lmp.flow	253M	100%	100%	13.94	177	187
CTH 2Gas	cth.2gas	500M	3%	16%	9.06	23	4268
CTH AMR	cth.amr	500M	<1%	8%	13.21	349	1575
CTH efp3d	cth.efp	500M	<1%	38%	11.83	146	7805
ITS brems cg	its.cg	415M	100%	100%	5.56	186	651
ITS brems CAD	its.cad	500M	20%	36%	5.77	184	449
sPPM	sppm	500M	<1%		17.95	5	53

Table 1. Benchmark Statistics for traces. Dhrystone and gcc.122 shown for comparison.

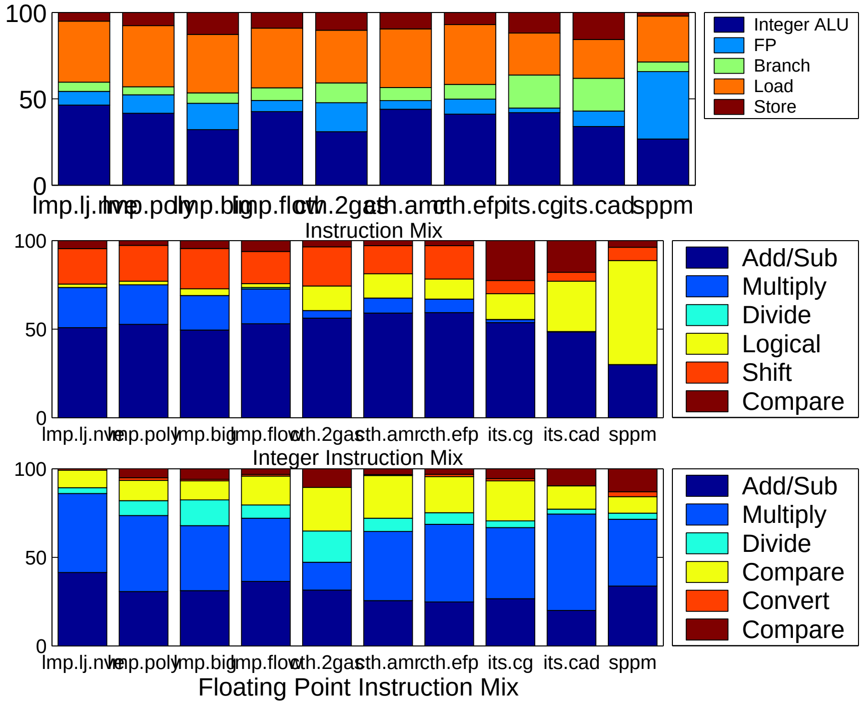

Figure 3. Benchmark instruction mixes

from different basic blocks cannot be executed in parallel.

An analysis of the traces reveals a significant potential for parallelism. Half of the benchmarks have a graph width of 6-8, and the remaining applications have a width of 8-12 (Figure 4). Once control dependencies are added in, much of this potential parallelism is reduced. Seven of the benchmarks fall in the 3-4 range. ITS, with its smaller basic block sizes, falls slightly below this and sPPM slightly above.

At the basic block level, all applications are dominated by basic blocks with widths in the 1-5 range (Figure 5). Basic blocks with widths greater than 10 are generally orders of magnitude less likely to appear. This distribution is reflected in the overall graph width with control dependencies for the applications, which tend to be less than 4 (Figure 4).

5. Results

In order for threadlets to be a viable computational paradigm, it must be possible to create threadlets that expose a significant amount of the inherent application parallelism without being overwhelmed by synchronization. This section explores the amount of parallelism that threadlets can extract from a basic block as well as the synchronization points between those threadlets. An analysis of the nature of those threadlets in terms of basic instruction types is also presented as a foundation for designing architectures that leverage threadlets.

5.1. Synchronization

The threadlet extraction process attempts to limit the number of values which are produced in one threadlet and consumed in another threadlet and thus require synchronization (see Section 3). It also tries to maximize the distance between data production and consumption, so values are computed earlier and consuming threadlets will not have to block as frequently. It is preferable to share a value that has already been produced, and will not require threadlets to block, than to have fewer dependencies that require more blocking. Though the average threadlet requires a number of values from other threadlets (Figure 6 and Table 2), many of these values will already have been computed. Additionally, because a number of threadlets may all consume the same value the number of *unique* values produced tends to be much lower than the number of times a value is shared (Figure 7). For example, in Figure 1 the main thread produces a value in register 1 that is consumed by two threadlets. Reuse of integer values is especially common. The average unique shared integer value is consumed by more than 3 threadlets (Table 2).

Threadlets that appear to produce or consume no data can be caused by a number of conditions. One artifact of the tracing process is that data that is generated outside of the trace (i.e. program input) or that is written but not consumed during the trace does not require synchronization. Threadlets that resolve branch instructions may not produce any data. Threadlets that use constants (i.e. load immediate) may not consume data and threadlets that perform "safety" initializations of data structures may produce data that is never consumed.

5.2. Threadlet Composition

As would be expected, threadlet lengths correspond closely to the target threadlet length (p) given to the partitioning algorithm (Table 2). Deviation from the target length is caused by the load balancing requirements.

The threadlets generated show a diverse range of compositions. Figure 8 shows a breakdown of the frequency of different compositions of threadlets for the LAMMPS lj.nve benchmark. The trends in the LAMMPS compositions were similar to those found in other benchmarks, so they have been averaged in Figure 9.

A number of threadlets contain no floating

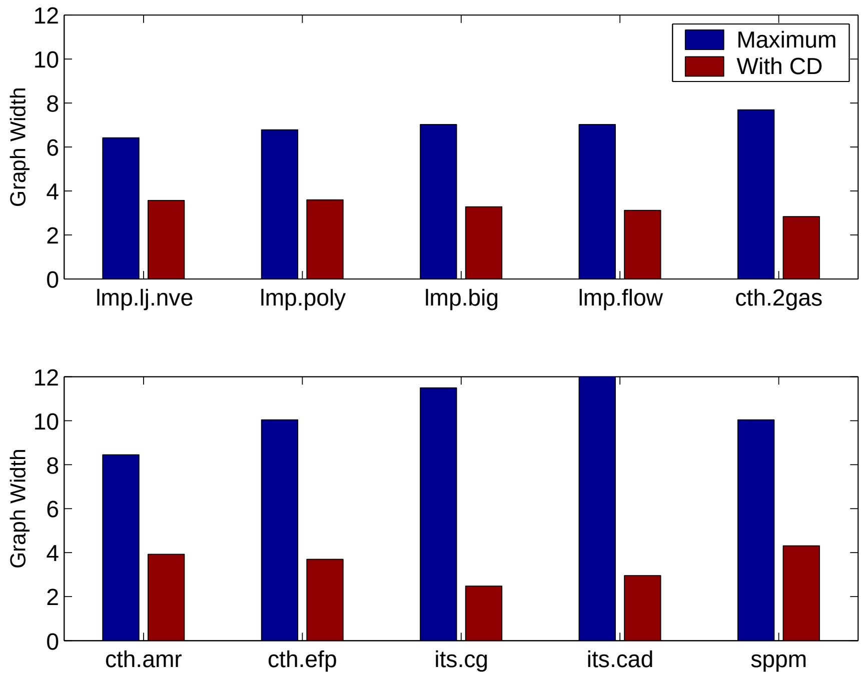

Figure 4. Data dependency graph width

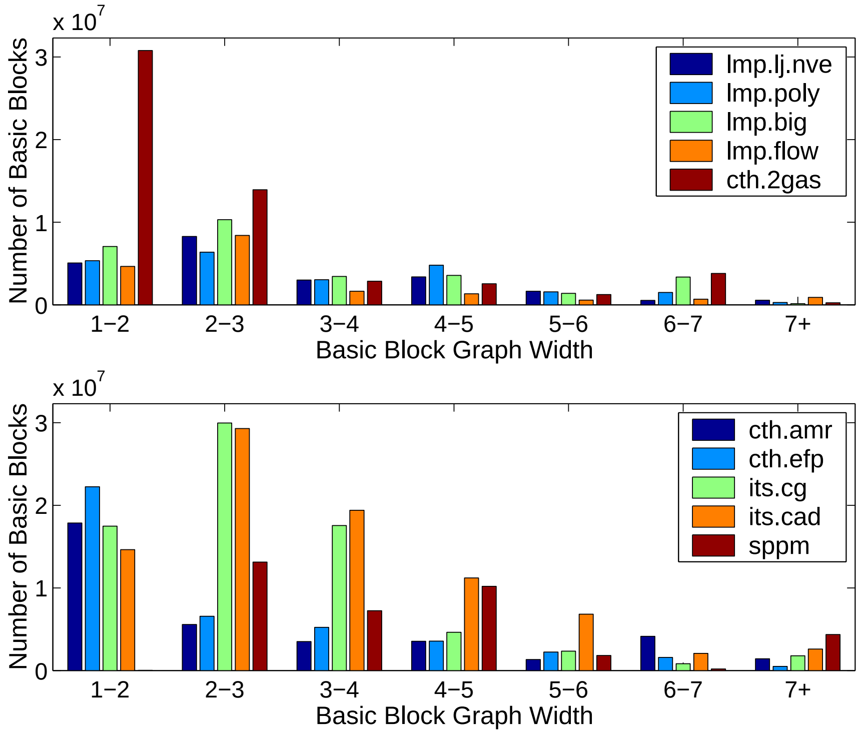

Figure 5. Basic block data dependency graph widths

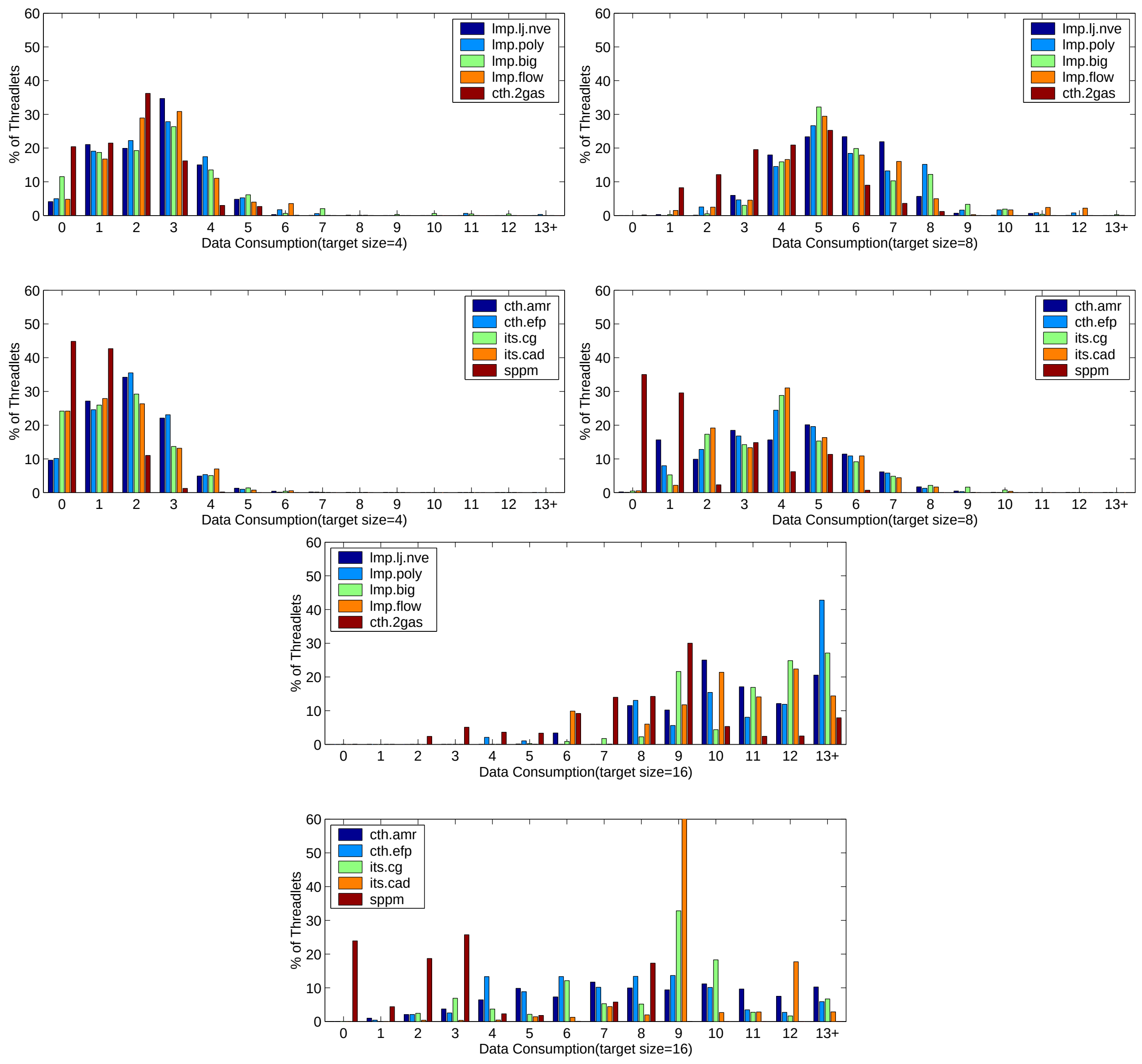

Figure 6. Histogram of values consumed per threadlet

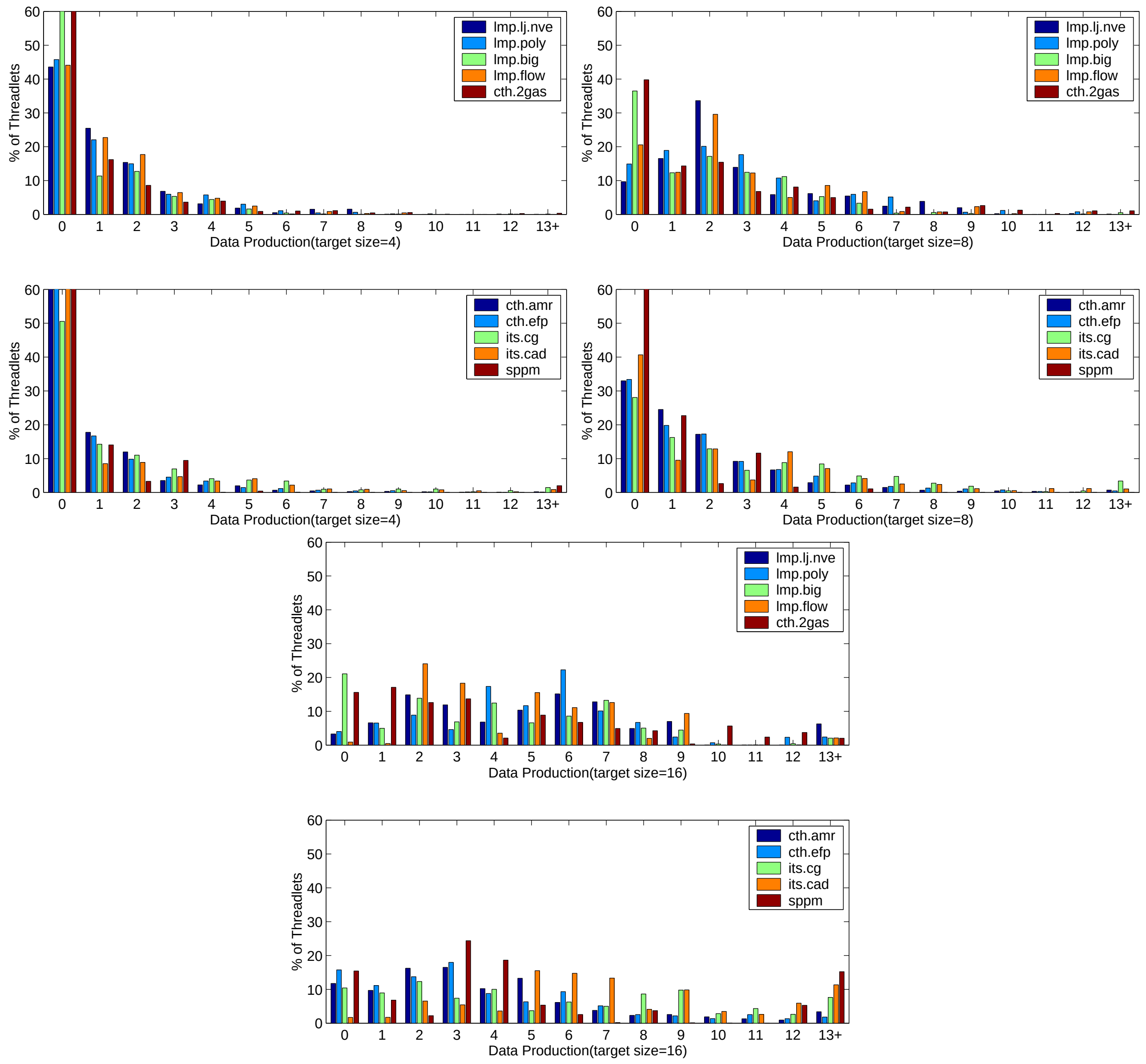

Figure 7. Histogram of values produced per threadlet

Measurement	Mean p=4	Mean p=8	Mean p=16	Std. Dev. p=4	Std. Dev. p=8	Std. Dev. p=16
Integer Instructions	1.73	3.76	7.83	1.43	2.05	3.44
FP Instructions	0.52	1.22	2.52	1.12	1.82	3.16
Memory instructions	1.41	3.21	6.74	1.36	1.74	3.19
Threadlet length	3.66	8.19	17.09	2.09	1.80	2.15
Integer Registers Produced	1.47	3.12	6.88	5.19	7.70	11.85
FP Registers Produced	0.23	0.49	0.98	1.25	1.62	2.15
Memory locations Produced	0.35	0.74	1.50	2.51	4.07	6.26
Total Values Produced	2.07	4.35	9.36	3.45	5.25	8.28
Integer Registers Consumed	2.01	4.52	9.38	1.47	2.17	3.71
FP Registers Consumed	0.31	0.69	1.55	0.86	1.24	2.02
Memory locations Consumed	0.59	1.34	2.75	0.83	1.41	2.44
Total Values Consumed	2.91	6.54	13.68	1.32	2.36	4.45
Unique Integer Values Produced	0.45	0.95	1.70	0.70	0.93	1.53
Unique FP Values Produced	0.16	0.29	0.48	0.41	0.57	0.82
Unique Memory Values Produced	0.34	0.74	1.48	0.61	1.02	1.52

Table 2. Statistics Summary. p indicates the target threadlet length for the extraction heuristic.

point operations, most likely due to the relative scarcity of floating point operations when compared to memory or integer instructions. Those threadlets which do contain floating point tend to mix floating point and memory instructions more than floating point and integer.

The most common class of threadlet comprises all memory or integer operations, usually in roughly equal proportions. Threadlets which are 50% memory and 50% integer account for 10% of all threadlets generated across the application set and parameter sizes. All memory and integer threadlets that have less than a 16% majority of integer or memory operations account for 61% of generated threadlets.

As the target threadlet size grows, the composition of threadlets becomes more diverse and we encounter more threadlets with different compositions. For example, when the target size is 4, only 9.2% of threadlets contain at least one floating point, integer, and memory operation. When the target size is 8, 17.9% of threadlets are mixed and when the target size is 16, 28.7% are mixed. This is because larger threadlets yield a greater number of possible permutations exist, and so a more diverse set of threadlet types can be extracted.

5.3. Parallelism Extracted

These threadlets manage to reveal a large portion of the available parallelism. We can compute the graph width of the programs after threadlet extraction in a manner similar to that used in Section 4.6. To find the widths of the threaded code, we assume that instructions within a threadlet must be executed sequentially, and that we have sufficient hardware to do so. The width of the graph of the extracted threadlets is a large portion of the maximum graph width with control dependencies (Figures 10 and 4). Smaller target threadlet sizes expose more parallelism because fewer instructions are serialized. With a target

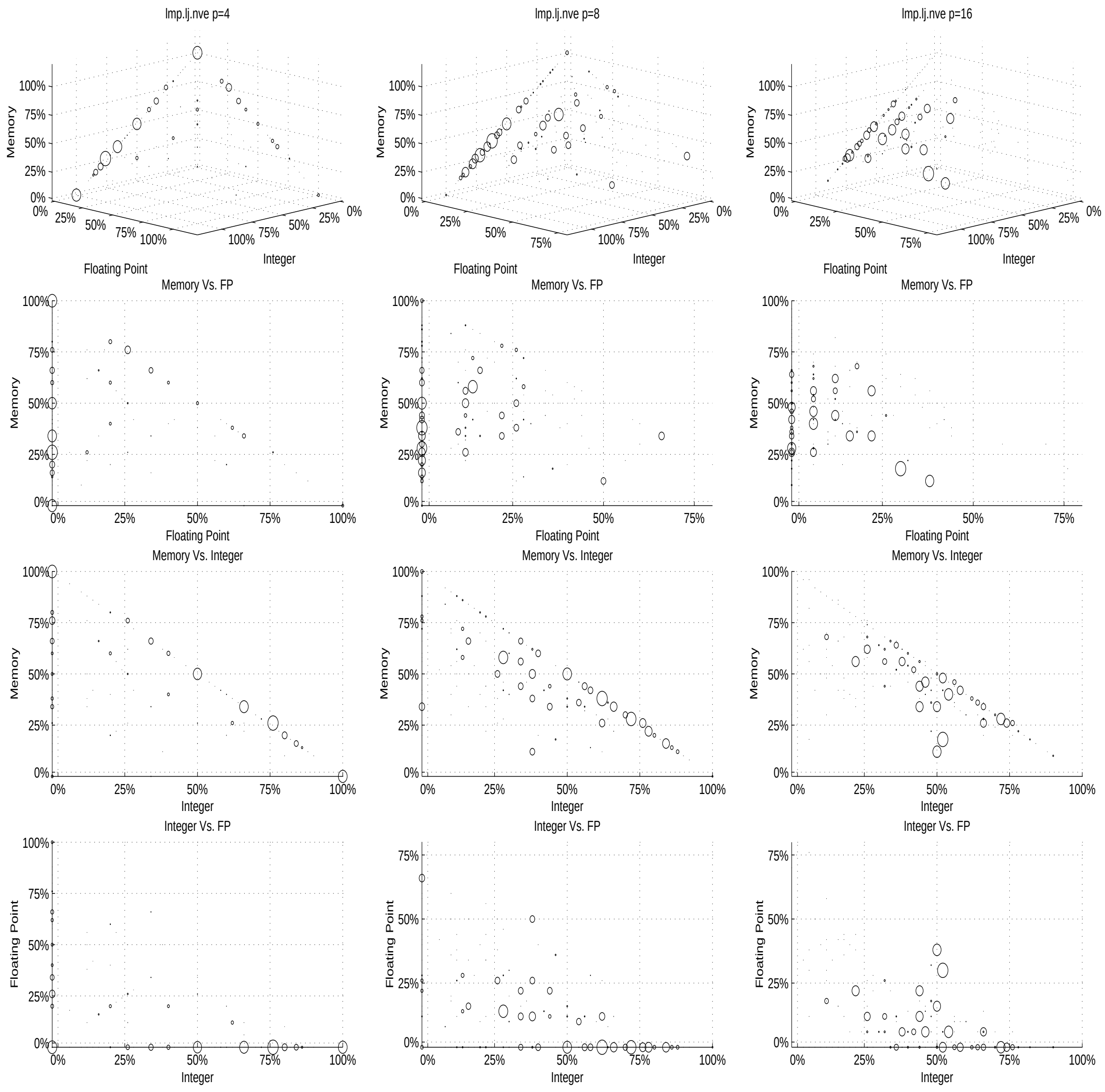

Figure 8. LAMMPS lj nve Threadlet Compositions: Size of the dot shows the frequency of threadlets of that composition appearing.

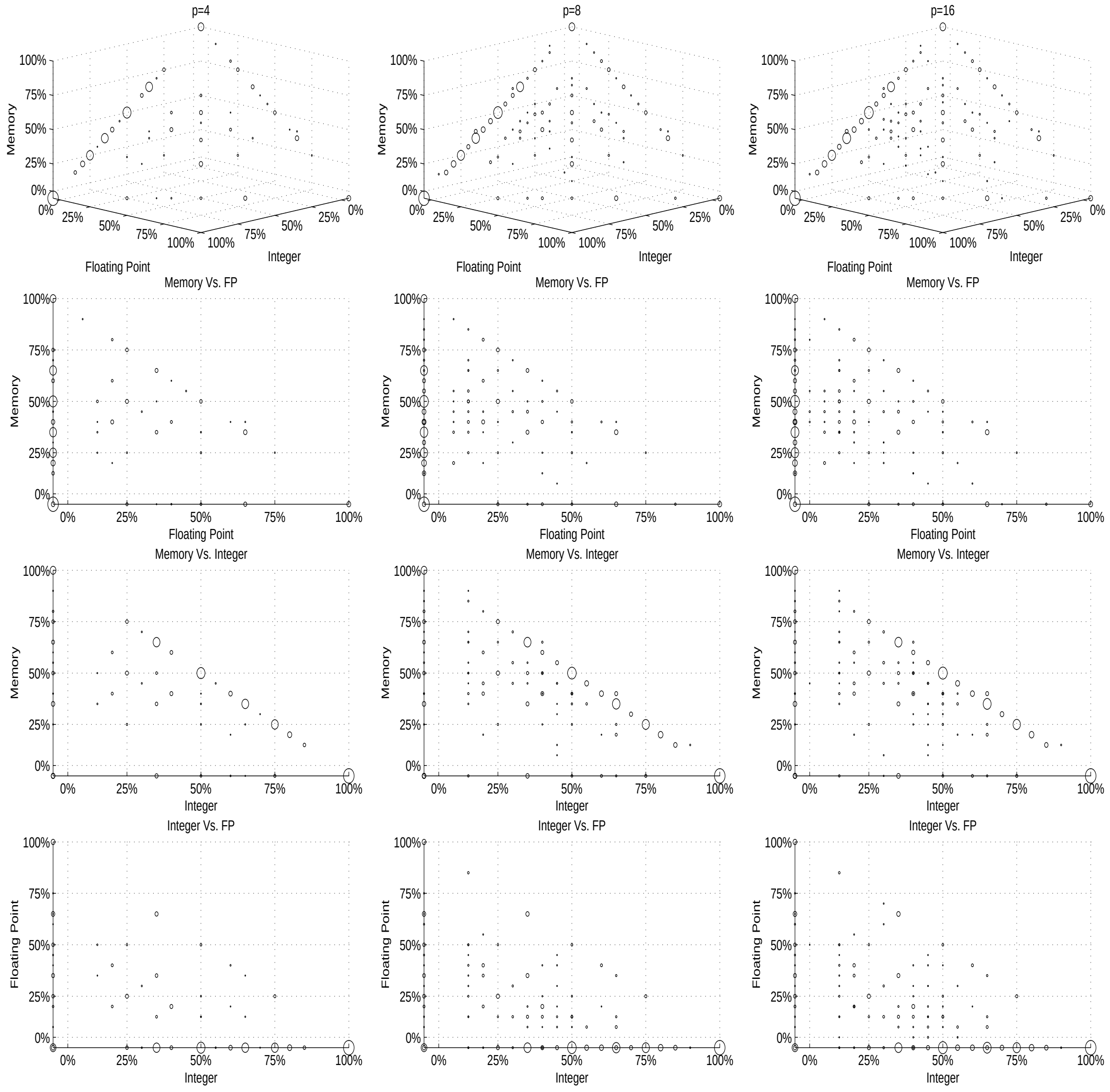

Figure 9. All threadlet compositions

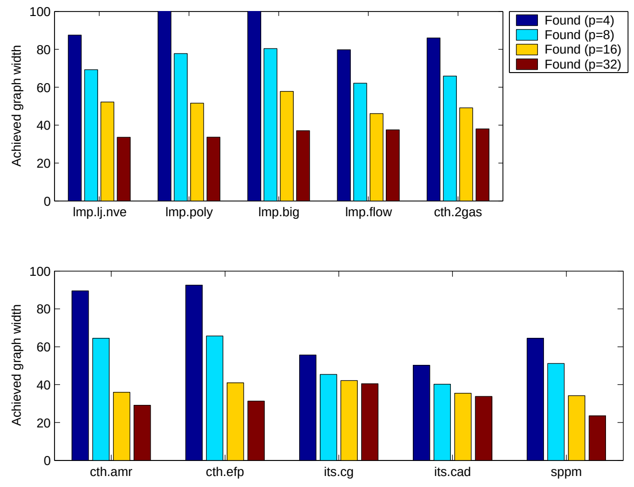

Figure 10. Percentage of average graph width, with control dependencies, achieved by steering-based threadlet extraction

threadlet size of four, 80% or more of the potential parallelism is revealed for seven of the 10 benchmarks. A target threadlet size of eight often achieves more than 60% of the unthreaded graph width. ITS and sPPM are exceptions to these trends. In ITS, this is due to a small basic block size which limits the potential for threading. sPPM has a higher than normal maximum graph width (with control dependencies) and it may be that the load balancing element of the threadlet extraction heuristic prevents this from being fully exploited.

6. Conclusions and Future Work

Scientific applications differ from the standard benchmarks that are used to evaluate processor architectures. They tend to have larger basic blocks and touch a larger amount of data during execution. This paper evaluates four such applications in the context of emerging support for multithreading in supercomputing architectures. Such support is slowly appearing in commodity microprocessors and is being taken to the next level in advanced architecture research funded through the DARPA HPCS program. The results present an interesting view into the ability to extract new levels of parallelism from important scientific applications through fine grain multithreading. In addition, the data indicates that conservative compiler approaches that only consider parallelism within a basic block could extract many of the fine grained threadlets.

This work will serve as a basis from which to develop new processor technologies and new supercomputer architectures. Going forward, this project will focus on using the data collected to architect PIM processors that can exploit the parallelism discovered, and as the basis for automatic threadlet extracting compilers. With further analysis, the data is expected to provide insights into the right microarchitectural decisions for a PIM and appropriate mechanisms for threadlet startup and inter-threadlet synchronization.

Acknowledgments

This work would not have been possible without the support and help of a number of individuals. Steve Plimpton (Sandia National Labs) provided LAMMPS, the appropriate input sets for it, and valuable assistance in describing those inputs. Brian Franke (Sandia National Labs) provided ITS, its input sets, and significant help in creating an appropriate description of the code and input sets. Sue Goudy (Sandia National Labs) provided valuable help in acquiring, compiling, running and selecting input sets for CTH. She also helped to track down descriptions of those input sets. Thanks also go to Jeffrey Vetter and John Engle from Lawrence Livermore National Labs for their help with sPPM.

The application analysis and data gathering discussed here was funded in part by Sandia National Laboratories, using tools funded in part by DARPA through Cray, Inc. under the HPCS Phase 1 program. The PIM architectures and ex-

ecution models are outgrowths of many previous projects funded by DARPA, JPL, and other agencies.

References

[1] Asci purple benchmark codes, July 2003. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html.

[2] Spatial corporate website, July 2003. http://www.spatial.com.

[3] sPPM general README file, July 2003. http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/sppm.readme.html.

[4] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeoung, G. D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, 1993.

[5] F. Allen, G. Almasi, W. Andreoni, D. Beece, B. J. Berne, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, P. Coteus, P. Crumley, A. Curioni, M. Denneau, W. Donath, M. Eleftheriou, B. Fitch, B. Fleischer, C. J. Georgiou, R. Germain, M. Giampapa, D. Gresh, M. Gupta, R. Haring, H. Ho, P. Hochschild, S. Hummel, T. Jonas, D. Lieber, G. Martyna, K. Maturu, J. Moreira, D. Newns, M. Newton, R. Philhower, T. Picunko, J. Pitera, M. Pitman, R. Rand, A. Royyuru, V. Salapura, A. Sanomiya, R. Shah, Y. Sham, S. Singh, M. Snir, F. Suits, R. Swetz, W. C. Swope, N. Vishnumurthy, T. J. C. Ward, H. Warren, and R. Zhou. Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer. *IBM Systems Journal*, 4(2), 2001.

[6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera System. *Tera Computer Company*.

[7] Apple Architecture Performance Groups. *Computer Hardware Understanding Development Tools 2.0 Reference Guide for MacOS X*. Apple Computer Inc, July 2002.

[8] J. Borkenhagen, R. Eickemeyer, R. Kalla, and S. Kunkel. A Multithreaded PowerPC Processor for Commercial Servers. *IBM Journal of Research and Development*, 44(6), November 2000.

[9] J. Brockman, P. Kogge, S. Thoziyoor, and E. Kang. Pim lite: On the road towards relentless multi-threading in massively parallel systems. Technical Report TR-03-01, Computer Science and Engineering Department, University of Notre Dame, 384 Fitzpatrick Hall, Notre Dame IN 46545, February 2003.

[10] J. B. Brockman, P. M. Kogge, V. Freeh, S. K. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ICS*, 1999.

[11] D. Burger. System-Level Implications of Processor-Memory Integration. *Proceedings of the 24th International Symposium on Computer Architecture*, June, 1997.

[12] J. B. Carter, W. C. Hsieh, L. Stoller, M. R. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA*, pages 70–79, 1999.

[13] L. Carter, J. Feo, and A. Snavely. Performance and programming experience on the tera mta, 1999.

[14] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. G. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *International Symposium on Microarchitecture*, pages 149–159, 1997.

[15] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Supercomputing, Portland, OR*, November 1999.

[16] G. Karypis and V. Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[17] P. M. Kogge. The execube approach to massively parallel processing. In *International Conference on Parallel Processing*, 1994.

[18] P. M. Kogge, J. B. Brockman, and V. Freeh. Processing-In-Memory Based Systems: Performance Evaluation Considerations. In *Workshop*

on Performance Analysis and its Impact on Design held in conjunction with ISCA, Barcelona, Spain, June 27-28, 1998.

[19] P. M. Kogge, J. B. Brockman, and V. W. Freeh. PIM Architectures to Support Petafbps Level Computation in the HTMT Machine. In *3rd International Workshop on Innovative Architectures, Maui High Performance Computer Center, Maui, HI*, November 1-3, 1999.

[20] M. S. Lam and R. P. Wilson. Limits of control fbw on parallelism. In *Nineteenth International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, 1992. ACM and IEEE Computer Society.

[21] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, February 2002.

[22] R. C. Murphy and P. M. Kogge. Trading Bandwidth for Latency: Managing Continuations Through a Carpet Bag Cache. In *Proceedings of the International Workshop on Innovative Architecture 2002 (IWIA02)*. IEEE Computer Society, January 10-11, 2002.

[23] R. C. Murphy, P. M. Kogge, and A. A. Rodrigues. The Characterization of Data Intensive Memory Workloads on Distributed PIM Systems. In *Proceedings of the Second Workshop on Intelligent Memory Systems, held in conjunction with ASPLOS-IX, Cambridge, MA*. ACM Press, November 12-15, 2000.

[24] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[25] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[26] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[27] S. J. Plimpton. Lammps web page, July 2003. http://www.cs.sandia.gov/ sjplimp/lammps.html.

[28] S. J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientifi c Computing*, Minneapolis, MN, Mar. 1997.

[29] J. Redstone, S. Eggers, and H. Levy. Mini-threads: Increasing tlp on small-scale smt processors. In *The Ninth International Symposium on High-Performance Computer Architecture (HPCA'03)*. IEEE, Feb 2003.

[30] A. Rodrigues. Rudra: A Reactive Dissipation Reducing Architecture. Master's thesis, University of Notre Dame, 2003.

[31] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 24, pages 290–302, New York, NY, 1989. ACM Press.

[32] T. Sterling and L. Bergman. A design analysis of a hybrid technology multithreaded architecture for petafbps scale computation. In *International Conference on Supercomputing, Rhodes, Greece*, June 20-25, 1999.

[33] T. Sunaga and e. a. Peter M. Kogge. A processor in memory chip for massively parallel embedded applicatiions. pages 1556–1559, October 1996.

[34] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fi ne-grained synchronization on a simultaneous multithreading processor. In *HPCA*, pages 54–58, 1999.

[35] V. Zyban. *Inherently Lower-Power High-Performance Superscalar Architectures*. PhD thesis, University of Notre Dame, March 2000.